Dedaub

Security Technology for Smart Contracts

DIFX Token

Smart Contract Security Assessment

Date: Jun. 14, 2021

Abstract

Dedaub was commissioned to perform a security audit of the DIFX Token (Difx). The Difx token is a simple ERC20 token, that is pre-minted according to a predefined supply. A large part of the minted tokens are sent into a vesting contract, which can be unvested by specific accounts over a number of months. Dedaub reviewed the code for 3 iterations. Following our suggestions and bug reports, the Difx token team re-architected the token logic and solved all the critical issues we flagged in the previous architecture.

This audit report is not to be construed as investment advice.

Setting and Caveats

We reviewed commit `d80e1b7f9ce2246ad383dff9fe629ce9f1cb4781` (after previous reviews of earlier commits and significant revisions as a result). Recommendations that led to the evolution of the previous versions were shared with the Difx token team. Several of them are listed in the report, but marked "**Resolved**".

Vulnerabilities and Functional Issues

This section details issues that affect the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
Critical	Can be profitably exploited by any knowledgeable third party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
High	Third party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
Medium	Examples: 1) User or system funds can be lost when third party systems misbehave. 2) DoS, under specific conditions. 3) Part of the functionality becomes unusable due to programming error.
Low	Examples: 1) Breaking important system invariants, but without apparent consequences. 2) Buggy functionality for trusted users where a workaround exists. 3) Security issues which may manifest when the system evolves.

Issue resolution includes "dismissed", by the client, or "resolved", per the auditors.

Critical Severity

Description	Status
Important variables such as development, `founders`, `coreTeam` are not getting set, they stay zero.	**Resolved**
`transfer` and `transferFrom` are infinite loops that will run out of gas, because they call themselves. The right solution is to call super.transfer/transferFrom, or to override the OpenZeppelin internal method `_beforeTokenTransfer`. The solution needs to be extensively tested.	**Resolved**
`transferFrom` doesn't have the override qualifier, seems to not even compile?	**Resolved**
The amounts of tokens minted does not fully agree with the tokenomics sheet supplied, nor even consistent with the rest of the code (e.g., `_strategicDevelopment` & `developmentAllowedSupply`).	**Resolved**
In `claimFounderReward/claimCoreTeamReward/claimDevelopmentReward`, the claimed supply against allowed supply check is too weak. A participant can claim an extra quarter's entitlement of tokens. This extra entitlement would deprive other participants from claiming their tokens.	**Resolved**
The logic for the `transfer` and `transferFrom` functions is broken, in several ways: • The check for amounts under vesting conditions is backwards: `if (oneYearVesting[msg.sender] <= amount) { ...` it should be a ">=". This logic appears in 4 places. • The logic of the if condition is wrong: `if (oneYearVesting[msg.sender] <= amount) { ... } else if (twoYearVesting[msg.sender] <= amount) { ... } else { return internalTransfer(recipient, amount); }`	**Resolved**

This code essentially places some conditions to limit the transfer of unvested tokens from specific accounts. It subsequently proceeds with the transfer with no restrictions if the conditions are not met.	
The `transfer` function is broken in how the tokens flow. The statement `internalTransfer(recipient, amount)` calls the default `transfer` of an OpenZeppelin ERC20 implementation, in the super-contract. This is wrong. It transfers tokens from `this` (the token contract itself) instead of from the sender. These tokens are minted to the `_privateSale` address in the constructor. Therefore this logic would transfer tokens from other categories of token holders (founders, core team, development), which are held in `this`. *(This had already been communicated in the earlier version of the code.)*	**Resolved**
It is generally a bad practice to confuse the `transfer` of tokens with the claiming of unvested tokens. An account that is entitled to unvested tokens could also have received tokens from a different account, bought them in the open market, etc. The check for vesting should not be part of the transfer, or it should be very careful to distinguish unvested tokens from vested. The current code effectively blacklists, forever, some accounts just because they are receiving non-vested tokens at deployment time: these accounts cannot be used normally. We recommend having a separate `claim` function instead of vesting restrictions over specific accounts in the `transfer` and `transferFrom` code. *(This had already been communicated in the earlier version of the code.)*	**Resolved**

High Severity

[No high severity issues]

Medium Severity

[No medium severity issues]

Low Severity

Description	Status
The time between vesting periods is 7776000 seconds. This is not technically a quarter of a year but 90 days.	**Resolved**
`transfer` and `transferFrom` always return false, so they're not ERC20 compliant	**Resolved**
Some fields are unused and/or uninitialized: • `founders` is uninitialized, but also unused • `coreTeam` is unused • `development` is unused • `foundersIndex`, `developmentIndex`, `coreTeamIndex` are internally unused. They could be made constant if kept, but probably should be removed.	**Open**
A number of storage variables are only set in the constructor and should therefore be marked as immutable	**Open**
The comments claim that privatesale tokens are vested but no logic in the contract enforces this.	**Open**

Other/Advisory Issues

This section details issues that are not thought to directly affect the functionality of the project, but we recommend addressing.

Description	Status
conditions like `oneYearVesting[msg.sender] == true` can be refactored to: `oneYearVesting[msg.sender]`	**Resolved**
There are many instances of “magic constants”: numeric constants (e.g., `7776000`) which should be better turned into constant fields, with names describing what these represent.	**Open**
The casts `IERC20(token)` in the Vesting contract are unnecessary.	**Open**
All functions in `Sale.sol` could be declared `external` instead of `public` since they are never called from within the contract. Although the parameters are not complex types it is still a good practice and makes the code more clear.	**Resolved**
Code in claim methods, such as:	**Resolved**

`` if (c) { transfer(recipient, amount); } else { revert("prohibited"); } ``` can be refactored to: ``` require(c, "prohibited"); transfer(recipient, amount); ```	
Use of a floating pragma: The floating pragma "`pragma solidity >=0.7.6`" is used, allowing the contracts to be compiled with any version of the Solidity compiler greater than or equal to 0.7.6. For major version updates, the code will not compile. Furthermore, even for 0.7.x versions, there will be small differences. Floating pragmas should be avoided and the pragma should be fixed to the version that will be used for the contracts' deployment.	**Resolved**
The Solidity compiler `v0.7.6`, at the time of writing, has known minor issues. We have reviewed the issues and do not believe them to affect the contract.	**Info**

Disclaimer

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness status of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, as well as a public bug bounty program.

About Dedaub

Dedaub offers technology and auditing services for smart contract security. The founders, Neville Grech and Yannis Smaragdakis, are top researchers in program analysis. Dedaub's smart contract technology is demonstrated in the contract-library.com service, which decompiles and performs security analyses on the full Ethereum blockchain.